Using clouds to scale grid resources: An economic model

Luis Rodero-Merino[a,b,*], Eddy Caron[a], Adrian Muresan[a], Frédéric Desprez[a]

[a] UMR CNRS - ENS de Lyon - INRIA - UCB Lyon 5668, 46 allée d'Italie, F-69364 Lyon, France
[b] Universidad Politécnica de Madrid, Facultad de Informática, B2 L3201, 28660 Boadilla del Monte, Spain

ARTICLE INFO

Keywords:
Clouds
Grids
Scalability
Economic model
GridSim
Fairness

ABSTRACT

Infrastructure as a Service clouds are a flexible and fast way to obtain (virtual) resources as demand varies. Grids, on the other hand, are middleware platforms able to combine resources from different administrative domains for task execution. Clouds can be used by grids as providers of devices such as virtual machines, so they only use the resources they need. But this requires grids to be able to decide when to allocate and release those resources. Here we introduce and analyze by simulations an economic mechanism (a) to set resource prices and (b) resolve when to scale resources depending on the users' demand. This system has a strong emphasis on fairness, so no user hinders the execution of other users' tasks by getting too many resources.

Our simulator is based on the well-known GridSim software for grid simulation, which we expand to simulate infrastructure clouds. The results show how the proposed system can successfully adapt the amount of allocated resources to the demand, while at the same time ensuring that resources are fairly shared among users.

1. Introduction

The term *cloud* [1] is applied to systems that allow to externalize the provision and management of computing resources. Infrastructure as a Service (IaaS) clouds supply virtual physical resources such as virtual machines (VMs). They are arguably the most well-known cloud type, and there are already several commercial offerings that provide such remote infrastructure services, e.g. Amazon EC2[1] or Rackspace.[2] The main feature of these cloud systems[3] is their ability to quickly supply virtual resources on demand, in commercial settings under a pay-per-use billing policy. Afterward, users can release those resources as soon as they do not need them. This way, users can handle peaks on capacity demand without incurring in resource overprovision. This characteristic is denoted as *scalability* and is getting a lot of attention from the research community [2]. Also, clouds provide a high degree of flexibility as each VM can use its own software stack. Hence, it is possible to run in the same physical host different applications even if they have conflicting software needs.

Grid systems, on the other hand, are a well-known technology that can provide a seemingly unique infrastructure from several resource providers, possibly heterogeneous. Typically, grid users send their tasks to the grid platform which will distribute them among the resources available. Activities such as resource location, execution scheduling, security handling, etc. are managed by the grid.

Grids can use clouds as infrastructure providers so they can deploy or release resources in order to react to changes on demand, or to anticipate to variations on that demand if load prediction systems (like [3]) are available. This demand of resources will be induced by the amount (which depends on the triggering rate) and size of tasks sent to the grid. Thus, grids will be able to allocate only the infrastructure they require. Besides, grids can benefit from clouds flexibility as they will be able to run tasks with heterogeneous software requirements in the same host. We deem this is of special interest in some typical grid usage scenarios where several users compete for resources which are freely (in monetary terms) available, but are also limited. Examples of such scenarios are several scientific environments, where resources can be provided by one or several entities. This proposal is mainly oriented to that kind of setups.

However, this brings a new problem: how can grids decide when to scale up or down resources? For example, a grid system could decide to enqueue incoming tasks, or even to reject them, instead of allocating new resources. Hence, it seems reasonable that users should be able to point out if their tasks have a certain priority so they should be run as soon as possible, instead of being enqueued or discarded.

* Corresponding author.
E-mail addresses: lrodero@fi.upm.es (L. Rodero-Merino), eddy.caron@ens-lyon.fr (E. Caron), adrian.muresan@ens-lyon.fr (A. Muresan), frederic.desprez@ens-lyon.fr (F. Desprez).

[1] http://aws.amazon.com/ec2.
[2] http://www.rackspace.com.
[3] From now on, the term *cloud* will be used to denote IaaS clouds.

Here we propose an economic mechanism to enable grids to decide how to scale resources. A price is computed for each resource, so the cost of running each task can be calculated. These prices are adapted depending on the demand. Users have a limited, periodically renewed budget to run their tasks. Negotiation follows a *tender/contract-net* model [4] where users ask for offers for each task they want to run and choose the most suitable one following a utility function also defined by them. The tender/contract-net model is known to be the economic model that optimizes users' utility [5], which is the main goal in the scenarios we address. Also, as no user can demand too many resources due to budget restrictions, no user can get a unfair share of those resources. Tasks have a deadline, so those that could not be run (not suitable offer was received) before their deadline expires will be marked as failed.

The main contribution of this work is the introduction of a hybrid grid–cloud architecture where one or more clouds provide infrastructure resources and the grid:

- Automatically scales resources usage to attend a variable demand to run tasks with possibly heterogeneous software needs.
- Splits resources fairly among users. Here, *fair* does not mean *equally*. Maybe some users need more resources than others, and those should be granted while there are enough resources for all.

In the presented architecture the grid system is not in charge of ordering users' tasks, which are processed following a FIFO policy. We assume instead that each user is the one who must prioritize her tasks following her own criteria, i.e.: the user is the one to decide which is the next task to execute. A tasks ordering mechanism for users is also proposed in this work, based not only on the priority assigned to each task by the user, but also on the risk of not being able to run that task which is computed using its size and the time to its deadline. This mechanism shows a better outcome than ordering tasks using only their priority. It is applied in the subsequent experiments to simulate a realistic dynamic market where users implement complex task management strategies.

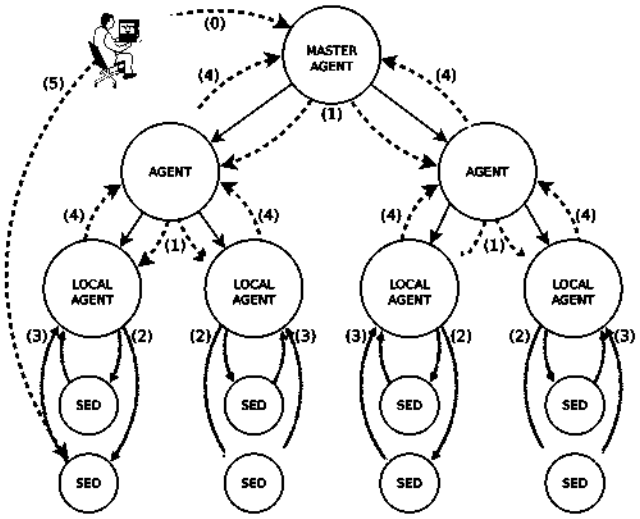
We test and evaluate this proposal by simulations run using the GridSim [6] simulator, whose features we extended in order to suit the requirements of our experiments. Experiments are run over a hybrid architecture that combines a grid system with IaaS clouds. The grid system used as basis of this architecture is DIET [7]. In [8] Caron et al. introduce and discuss a first proposal of the architecture presented here.

The rest of the paper is organized as follows: Section 2 details the architecture proposed; Section 3 explains how the system market approach is implemented, i.e. how currency flows, how offers for each task request are computed, how prices are adapted, etc.; Section 4 shows the results of some simulations that test key features of the proposed system; Section 5 presents an analysis of related work in the area of clouds and economy-based grid systems; finally Section 6 discusses the conclusions of the work presented. The extensions proposed to GridSim, which were the base for the simulations presented in Section 4 are briefly explained in Appendix.

2. Grid–cloud architecture

The solution proposed in this paper combines a hierarchical grid system, DIET, with several clouds that will provide resources to the grid. To describe this solution we need first to outline how DIET works.

DIET [7] connects its components through a hierarchical tree for scalability. The basic DIET component is the *Agent*. Agents have scheduling and data management capabilities, but here we will focus on their primary and most basic functionality: service location. Fig. 1 depicts DIET components organization. Each DIET grid has a unique *Master Agent* (MA) on the top of its hierarchy. This MA gets service requests from users. Each request



(0) The user issues a new task request to the Master Agent.
(1) The user task is forwarded down through DIET hierarchy.
(2) Finally the request reaches the SeDs at the bottom.
(3) The SeD buids an answer, taking into account its own capacity. It forwards it to the parent Local Agent.
(4) A list of SeDs able to fulfill the request Offers is sent back through the hierarchy to the Master Agent who forwards it to the user. That list is ordered by SeDs' capacities.
(5) User chooses the best SeD and sends it the task.

Fig. 1. DIET hierarchical layout.

goes down through the hierarchy formed by the agents until it reaches the *Server Daemons* (SeD), that interact with the execution environments and provide the actual execution services. Each Agent knows the services that can be executed by the SeDs at the bottom of each one of its children Agents, and it will not forward service requests to those Agents whose corresponding SeDs cannot run the service. Each SeD is connected to DIET's hierarchy through *Local Agents* (LA), LAs are intended to be at the resources provider site. When some request reaches the SeD, it builds a reply reporting its state. Replies are sent back through the hierarchy up to the MA. Replies are ordered by some objective function that depends on the SeDs' state, so the "best" SeDs are first in the list. Finally the MA will send the list of replies to the user, who will pick some SeD in the list (usually the first one) and command it the task to run.

DIET's layout makes straightforward to connect IaaS clouds as resource providers to the grid. IaaS systems will be connected to the SeD nodes, who will decide when to scale (allocate and release) resources to attend users requests. Services will be run in the VMs hosted in the cloud. IaaS providers can be built on top of hardware providers by using several open solutions such as OpenNebula [9], Eucalyptus [10] or Nimbus [11]. Such solutions have simple remote interfaces that SeD nodes can use to request the creation of VMs and/or networks to connect them. Once a VM is created, the SeD node will be in charge of connecting to it to run services in order to attend users' tasks. Fig. 2 shows a first sketch of the elements involved in the described layout, using OpenNebula as a possible IaaS Provider. In our proposal the user interacts at all times with DIET elements (MA and SeDs). She is totally unaware about the fact that SeDs may run tasks in VMs supplied by IaaS clouds.

The hybrid approach presented here is detailed in Fig. 3. A new module for task allocation is placed between the IaaS system and DIET's SeD node (Task Allocation Module, TAM), that will be in charge of computing where the tasks sent by users can be executed and will adapt prices as demand changes. A task can be run in an already active VM, or in a new VM that will be demanded by the TAM to the cloud provider. The cloud provider

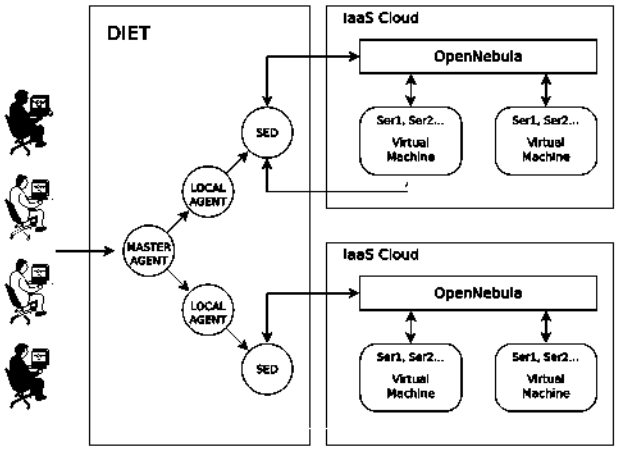


Fig. 2. Sketch of the proposed hybrid grid–cloud layout.

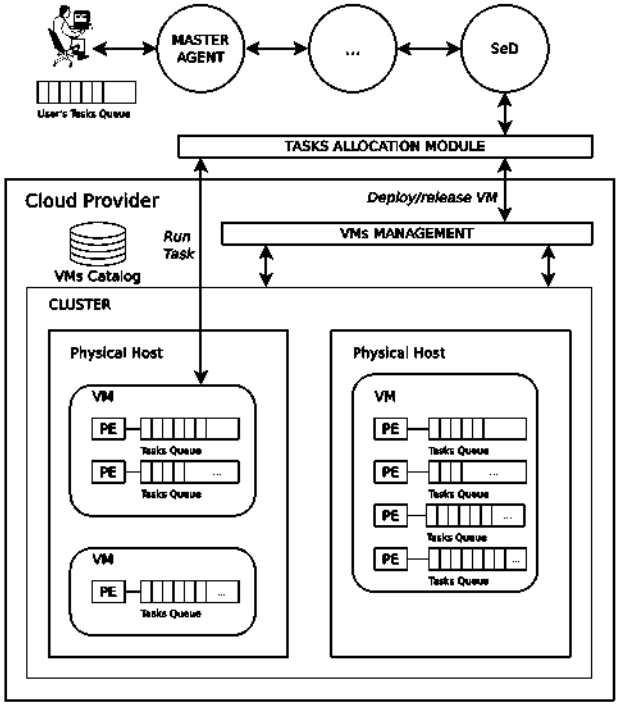


Fig. 3. Architecture overview.

will have a catalog describing the hardware configuration of the VMs that the TAM can instantiate. Each VM will have one or more processing elements (virtual CPUs) with their corresponding queue of pending tasks. When computing allocations for a given task, the TAM must take into account the tasks already in the queues of each VM. The TAM can ask to the cloud provider whether a VM of a certain type can be instantiated or not which will depend on the resources of the physical hosts available. This is necessary so the TAM can determine allocations in new VMs. We assume that a set of disk images containing the VMs software stack (OS, libraries,...) required to run the tasks is available.

Now, the main goal of the grid system is to ensure that resources are shared in a fairly manner. To achieve this we propose a market-based approach, that is described in detail in the next section. The characteristics of this approach impose certain changes in the way SeD nodes run. Those changes are also explained in the next section.

3. Using markets to reach fairness

Markets can be defined as a way to exchange "goods", in this case the right to run tasks on some infrastructure. In such market, resources have a certain price associated, and so users must take into account their (limited) budget to decide when and where to demand the execution of those tasks. If resource prices are set taking into account the demand, and budgets are allocated equally among users, by intrinsic market dynamics we can expect resources to be fairly shared (a more thorough discussion about the role of markets as a solution for fair resource sharing can be found in [12]).

When designing a market environment several decisions must be taken regarding different features:

- How currency is managed.
- How negotiation is performed, i.e. how requests are sent and how offers are collected.
- How offers for each user request are built.
- How resource prices, that determine each offer cost, are computed.
- How the user chooses the best offer.

The rest of this section describes the characteristics of our proposed market and explains the design decisions taken regarding them.

3.1. Currency

Users budget will be bounded by the amount of *virtual currency* they have (using real money is possible, but is has several drawbacks, see Section 5.1.2). An initial budget is assigned to each new user in the system. Users cannot run tasks beyond their budget. On the other hand, currency should be assigned to users to avoid the potential problems of *starvation* (users cannot access resources), *depletion* (users hoard currency to monopolize resource access at certain times) and *inflation* (prices grow due to uncontrolled addition of currency to the system) [13]. Several options are possible:

- The global value of all resources is periodically computed, taking into account their present prices. This would represent the total "wealth" of the system. This amount is then split and sent to the users.
- A given fixed amount is sent periodically to all users. Providers (i.e. clouds) do not hoard the money they get from users.
- SeD/Clouds do not hoard neither drop the money received from the users. Instead, all that money is periodically gathered and forwarded back to the users.

The two first options can easily lead to inflation as currency is injected to the system even if the demand of resources is low. Also, new users will be in an adverse situation as previous users can hold big amounts of virtual currency. Thus, the third option seems the more feasible, and is in fact similar to the idea proposed in Mirage [14] (see Section 5). Our proposal adds a new entity, the *Virtual Bank*, which will be in charge of gathering all the incomes of the cloud providers. Periodically, the currency collected by the Virtual Bank is evenly split and sent to the users. Payments from users to providers are done directly once the corresponding task execution is finished, with no intervention from the Virtual Bank.

3.2. Task execution negotiation

Every time the user needs to run a task, it sends a REQUEST_FOR_OFFERS message that through DIET hierarchy will reach all available SeD nodes (in fact, their corresponding TAM modules, see

Fig. 3) connected to some cloud provider. Our simulations take into account three resources (more can be easily added): CPU, disk and memory. Hence, each request contains information about the required amount of resources (CPU measured in MIs, memory measured in MBs, and disk measured in GBs).

When a REQUEST_FOR_OFFERS message reaches a certain TAM, this module will build a set of offers to execute the task. The process of creating offers for a request is detailed in Section 3.3. An allocation offer A is a tuple that contains the cost and time that it will take to run a given task $(A^{\text{TIME}}, A^{\text{COST}})$. A TAM can create none, one or many allocation offers for a task T_i. When all possible allocations to run the task have been computed by the provider they are sent back to the user in an OFFERS message. If the provider could not find any suitable offer then the message will be empty. OFFER messages are sent again through DIET. Each node in the hierarchy (LAs, Agents and the MA) will gather all the offers they receive from their children nodes for each REQUEST_FOR_OFFERS they had forwarded before, and will build a new OFFERS message with the offers carried by the OFFERS messages from its children. Of course, the node will not build and send the new OFFERS message for a task until it node has received an OFFERS message for that task from all its children.

Finally, only one OFFERS message will reach the user, containing all offers from all SeDs. Then, the user will choose the most suitable allocation offer using some utility function, and will send a RUN_TASK message directly to the corresponding provider. If the OFFERS message is empty, or it does not contain any suitable offer, then the task is stored in a queue by the user to be tried again later. An offer is not suitable for a task if its cost A^{COST} is greater than the available user's budget, or if the time to execute it A^{TIME} exceeds the task deadline. Each user periodically checks the tasks stored, discarding as failed those tasks whose deadline has expired.

A RUN_TASK message carries the time and cost conditions from the chosen offer. When the TAM receives such message, it computes again possible allocations for the task to check if it still can honor the offer. If it is not so (due to shortage of resources or changes on resource pricing) then the user is notified. In such case the task is stored by the user as if it had no offer. If the task can still be run under the offer conditions then it is executed. When the task is finished the result is sent to the user by a RUN_RESULT message, which carries the task results or the corresponding errors. If the task could not be run due to some reason (e.g. unexpectedly the deadline was surpassed during execution) then the user discards the task as failed.

3.3. Building tasks allocations offers

Before describing how offers are built by cloud providers, it is necessary to outline how physical hosts and VMs are characterized. Then we describe the process of computing all possible options to run a task. Each option will then become an allocation offer $(A^{\text{TIME}}, A^{\text{COST}})$ that will be sent in the corresponding OFFERS message.

3.3.1. Physical hosts and virtual machines

Each cloud provider has a *catalog* of VM types available $\{V_1, \ldots, V_n\}$. Each VM type V_j defines a hardware configuration with the resources it has: amount of Processing Elements PEs[4] V_j^C and their processing speed V_j^s (in MIPS); memory V_j^m; and disk V_j^d. Also, there is information about how long it takes to start a VM of that type V_j^{START} and the price of creating such instance V_j^{COST}. Each cloud provider has a set of m physical hosts $\{H_1, \ldots, H_m\}$. Each host H_k has a set of H_k^C CPUs all with the same processing capacity H_k^s. For each processor $p_{k,l}$ $(1 \leq l \leq H_k^C)$ in host H_k we represent by $p_{k,l}^a$ the available processing capacity of that CPU (in MIPS), i.e. the processing capacity not used for any of the VMs allocated in the host. Conversely $p_{k,l}^u$ is the used capacity, so $p_{k,l}^a + p_{k,l}^u = H_k^s$. Also, the amount of memory in host k is given by H_k^m, while $H_k^{m,a}$ and $H_k^{m,u}$ are the available and used memory in that host respectively. H_k^D represents the amount of disks of host k, and H_k^d represents their capacity. For each disk $z_{k,l}$ $(1 \leq l \leq H_k^D)$, $z_{k,l}^a$ and $z_{k,l}^u$ are the available and used storage capacity of disk $z_{k,l}$ $(z_{k,l}^a + z_{k,l}^u = z_{k,l})$.

When a new VM of type V_j is allocated in some host H_k then the corresponding values are updated. The PEs must be allocated in V_j^C physical CPUs (of course $V_j^C \leq H_k^C$) with enough available capacity. For example, processor $p_{k,1}$ would be assigned one of VM's PEs only if $p_{k,1}^a \geq V_j^s$. When one PE is assigned to some physical CPU its corresponding parameters are updated so for example $p_{k,1}^a = p_{k,1}^a - V_j^s$. Also, the host available memory must be enough to allocate the VM memory, and if so then it must be updated when the VM is finally created $H_k^{m,a} = H_k^{m,a} - V_j^m$. Finally, the capacity of the disk where the VM storage will be set is also updated so $z_{k,l}^a = z_{k,l}^a - V_j^d$. $H_k^{m,u}$ and $z_{k,l}^u$ are updated likewise. If there are more than one host where the VM can be created, then the host running more VMs is used for the new VM. The goal is to use as less physical hosts as possible at all times, which in turn should impact on the power consumption (unused hosts can be in sleep mode, which will demand less power). On the other hand, as time passes some VMs can become *idle*, i.e. they have run all tasks assigned and are waiting for new tasks to be executed. Periodically it is checked how long each one of these idle VMs has been in that state. If any VM has been idle for a period longer than a certain threshold time, that VM is switch off and its resources are released. So if the VM was of type V_j and was running on host H_k, then the available resources are updated as expected: $H_k^{m,a} = H_k^{m,a} + V_j^m$, and so on.

All PEs have a FIFO queue of tasks associated. When a RUN_TASK message reaches a cloud provider the set of possible allocations must be computed again to check whether that task can be run within the cost and time originally offered (which are carried by the RUN_TASK message). If so, the provider will choose among the found allocations the one that maximizes the user's utility function. Depending on the allocation, the task can (1) be assigned to a free PE and start immediately; (2) be assigned to a PE that is busy (an then it will be added to the PE's task queue); (3) require to start a new VM, in such case a new VM instance will be created, once it is ready the task will be assigned to any of its PEs. The algorithm to compute all possible allocations for a task is described in the next section.

3.3.2. Task allocations computation

An allocation for a task is the assignation of the task to a certain PE in some VM. Each allocation will have a cost and duration $(A^{\text{TIME}}, A^{\text{COST}})$.

When an user asks for offers to compute a task, or sends a request to execute it, potential allocations for that task must be looked for. In the former case, each allocation found is sent back to the user as an offer (see Section 3.2). In the latter case, if some of the possible task allocations meets the time and budget given by the user, then the task will be processed by the corresponding PE.

All the possible allocations for a task are calculated by an algorithm that comprises two steps: (1) first the TAM analyzes the VMs already present and whether they can run the task; (2) then the possibility of creating new VMs to run the tasks is checked. The output of each step will be a collection of allocations. Both sets will be combined resulting in the final set of potential allocations for the task. The remaining of this section details these two steps, specifying also how A^{COST} and A^{TIME} are computed for each allocation:

[4] To avoid confusion with physical CPUs, we will denote as PEs the VMs' CPUs.

1. First, the TAM analyzes the state of the already present VMs in order to find running VMs where the task could be executed. They are grouped by the VM type (V_j) they belong to. These VMs can be active (running some other tasks) or idle (all PEs are free). Idle VMs are checked first.
   For a task i, let c_i, d_i and m_i be the amount of CPU, memory and disk required by that task respectively. The time to run the task i in an idle machine of type V_j is $A^{\text{TIME}} = c_i/V_j^s$. Regarding cost computation, let P_m, P_d and P_c be the price of 1 MB of memory, 1 GB of disk, or the computation of MI (prices computation is explained in Section 3.4), then the cost of the task is computed as:

$$A^{\text{COST}} = P_m m_i + P_d d_i + P_c c_i. \tag{1}$$

   After looking for allocations in the idle VMs, active VMs are checked too, i.e. those VMs whose PEs are running some other tasks. For each active VM of type V_j, the TAM checks each one of its V_j^C PEs to see when it will be available (it will not be running any task and its queues are empty). Let q be the amount of tasks waiting in the PE's queue, numbered from 1 to q. Let $\{c_1, \ldots, c_q\}$, $\{m_1, \ldots, m_q\}$ and $\{d_1, \ldots, d_q\}$ the CPU, memory and disk those tasks demand. Let also c_0 the remaining MIs to be executed of the task being run when the allocations are computed. Then, the PE will be busy until $t_b = \left(c_0 + \sum_{0<x\leq q} c_x\right)/V_j^s$. If at t_b the amount of disk and memory that will be available in the VM (i.e., not used by the tasks run by the others PEs in at t_b, which is known studying their queues) will be enough to run the task, then a new allocation where the task is assigned to that PE can be built. The time to run the task will be:

$$A^{\text{TIME}} = \frac{c_0 + \sum_{0<x\leq q} c_x + c_i}{V_j^s}. \tag{2}$$

   The cost of running the task is computed as before (see Eq. (1)).
2. Second, each VM type V_j is analyzed to check (a) if a *new* VM instance of that type could run the task, i.e: $V_j^m \geq m_i$; $V_j^d \geq d_i$; (b) if there is any physical host H_k with enough spare capacity where the VM can be instantiated, that is, it has enough available memory $V_j^m \leq H_k^{m,a}$, it has some disk with enough available storage $V_j^d \leq z_{k,t}^a$ and it has V_j^C processors with enough spare processing capacity V_j^s.
   If both conditions are met, then a new allocation has been found. The allocation time is computed as the addition of the time to start the VM, plus the time to run the task itself:

$$A^{\text{TIME}} = V_j^{\text{START}} + \frac{c_i}{V_j^s}. \tag{3}$$

   The allocation cost is computed as the addition of the cost of instantiating the VM, plus the cost of using the resources for the duration of the task which depends on their price. Let P_m, P_d and P_c be the prices of memory, disk and CPU (price computation is explained in Section 3.4). Then:

$$A^{\text{COST}} = V_j^{\text{COST}} + P_m m_i + P_d d_i + P_c c_i. \tag{4}$$

3.3.3. Choosing the best allocation to run a task

When the SeD receives a task to run (in a RUN_TASK message) and the TAM has computed all the suitable allocations for that task, then one of those allocations must be chosen. The TAM applies the user' utility function to choose which is the best allocation choice.

But in some cases different allocations will have the same time and cost (and so the same utility value). For example, one allocation can run the task in an already active VM with some free PEs, and another one can run the task in an idle VM of the same type. So, when several allocations have the same time and cost the TAM applies some heuristics that favor energy saving to choose the definitive allocation for the task:

- The grid will prioritize those allocations that will run the task in an already active VM (i.e. one or more of its PEs are running tasks).
- If no allocation in an active VM is found, then the grid will prioritize those allocations that assign the task to an already present VM (which will be idle). If there are several idle machines, the VMs that have been idle for the shortest period of time are preferred. The goal is to keep idle machines in that state while possible, so their resources will be eventually freed when they are shut down (the grid shuts down the VMs that have been idle for longer than a certain time).
- Only if no allocations in active or idle VMs are found, then allocations that require instantiating a new VM are considered.

3.4. Resource prices computation

The price adaptation mechanism applied takes into account the resources demand to change prices accordingly. This algorithm is run periodically by the TAM to compute the price of the resources in the cloud.

A cloud provider will price resources differently depending on the goals pursued. To maximize benefits, the provider could apply the approach explained in [15]. But in collaborative environments, the cloud provider can also try to maximize resource usage and so the amount of tasks run. The latter is the approach taken in this work.

The algorithm goes as follows. Let r be the total amount of some resource in the provider's site, measured in a certain unit (e.g. MBs of memory). Let $r^d(t)$, $r^a(t)$ and $r^w(t)$ the amount of demanded resources by all tasks in the grid (running or in queues), available resources, and resources demanded only by waiting tasks at time t. The amount of free resources $r^a(t)$ is given by the addition of the free resources in all physical hosts plus the available resources in all the virtual machines they run (i.e. unused by the tasks being processed at that moment). At all times $r^d(t) = r - r^a(t) + r^w(t)$. Also, often (but not always) if $r^w(t) > 0$ then $r^a(t) = 0$. Let $P_r(t)$ by the price at time t. Price is adapted periodically every s seconds as described in Eq. (5) (let $t' = t + s$):

$$P_r(t') = \begin{cases} P_r(t) \times \left(1 + \dfrac{r^d(t') - r}{r}\right)^{\frac{r^d(t')}{r^d(t)}} \\ \quad \text{if } r^d(t') > 0 \wedge r^d(t) > 0 \\ P_r(t) \times \left(1 + \dfrac{r^d(t') - r}{r}\right) \\ \quad \text{if } r^d(t') > 0 \wedge r^d(t) = 0 \\ P_r(t)/2 \qquad \text{if } r^d(t') = 0. \end{cases} \tag{5}$$

The first case in Eq. (5) aims to increase (decrease) the price depending on the amount of resources demanded over (below) the total available. The exponent modulates the adaptation depending on how sharp the change on resources demand has been since the last price recomputation. The second case is identical to the first one, to be applied when $r^d(t) = 0$. Finally, if the amount of resources used at t is 0, then the price is divided by 2.

3.5. Processing of offers by users

To simulate real users behavior is far from trivial. Usually, logs of task requests in real-world grids are helpful to reproduce a real load. However, they do not capture users reactions to situations where their requests could not be run due for example to resource contention, i.e. how they prioritize their tasks, how they choose between different execution options from different grids when

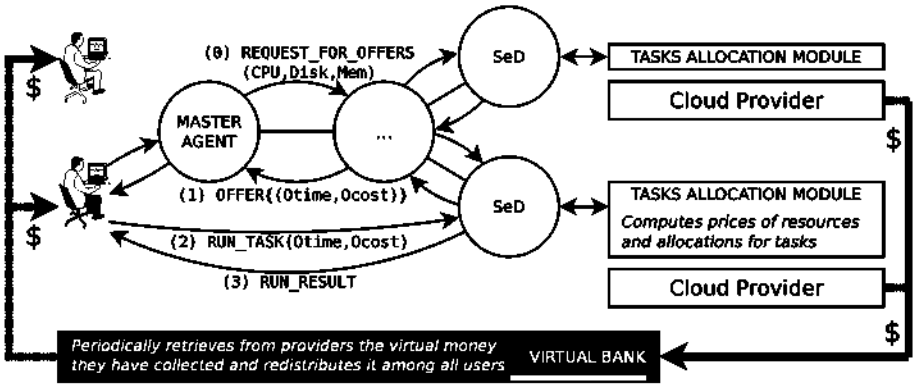


Fig. 4. Main architecture elements and interactions.

available (usually grid usage logs refer to a single grid), how many tasks were outdated while waiting in the users queues, etc. Thus, instead of going through static load records, we chose to simulate users as dynamic entities that take planning decisions about their tasks.

3.5.1. Utility function for offer selection

For each REQUEST_FOR_OFFERS issued by the user, the MA will send back a list of possible allocations (offers). The user will first filter those offers that cannot be accepted because of time or cost restrictions. Each task has a deadline associated, so offers that would last beyond that deadline will not be considered by the user. Also, if the offer cost is greater than the user actual budget the offer is likewise rejected.

Then, the user must choose the best offer among the remaining ones. This depends on the user own priorities. Users will define a *utility function* $u : \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \to \mathbb{R}$ to express the "utility" or worth of each task depending on the cost and time to execute it. The utility function is applied to the offers received, and the offer with the greatest utility value will be chosen. A possible utility function is $u(A^{\text{COST}}, A^{\text{TIME}}) = (A^{\text{COST}} \times A^{\text{TIME}})^{-1}$. If the user is only concerned about the time to execute the task regardless of its cost then the utility could be defined as $u(A^{\text{COST}}, A^{\text{TIME}}) = (A^{\text{TIME}})^{-1}$. The goal is to enable users to express their preferences, e.g. not to spend too much in a task (although it takes longer to run it) or to run the task as quickly as possible (even if it is expensive).

3.5.2. Negotiation strategies

When some user requests execution offers for a task, she can face different situations:

- No offer is received, or all fail to meet the time and cost bounds imposed, that is the task deadline and the user budget. Then, the user can just label the task as "failed" or store it in a queue for later retrial.
- Some offers are suitable. Then, the best is chosen using the utility function as described in 3.5.1. The task is sent to the corresponding provider. In such case, still two things can happen.
  - The offer can still be honored by the provider, and so the task is executed.
  - The provider cannot fulfill the offer any more (e.g. due to a load burst after the offer was computed). Again, the user can then ignore the task or retry it.

If failed tasks are stored, then users will prioritize them to set which tasks must be tried again first. Each task i will have a value I_i to represent its importance/priority. A basic strategy is to order tasks by importance so those with higher I_i values are retrieved from the queue and tried again first. We denote this strategy *Priority by Importance*. Yet, in real situations users will take also into account the "risk" of not being able to execute some task before its deadline expires. For example, if one task has a higher importance than another one, but there is still plenty of time to run the first while the second task's deadline is close, then the user can prefer to run the second task first. We propose the following mechanism for our simulations to set the tasks priorities: for each task i we compute the risk of not being able to run it on time as the coefficient between the task size (c_i) and the remaining time until the task deadline T_i (how the deadline is computed is explained in Section 4), which at time t is $T_i - t$. Then, this risk is multiplied by the task importance I_i to get the priority of the task. So, the priority of task i at time t is computed as $c_i \times (T_i - t)^{-1} \times I_i$. The user queue that stores the tasks will order them by this value. We denote this other strategy *Priority by Risk*.

Users will periodically check if they have pending tasks stored, choose the one with the highest priority, and start the negotiation to run the task (see Section 3.2). This is done also every time that the user receives the result of another task.

Also, when the user has chosen the best offer for a certain task, she can store the other suitable offers instead of just discarding them. Thus, if the offer initially chosen is not valid anymore, then the user can try the other alternative offers before requesting new ones. In that case, providers can also send alternative offers when they are not able to run the task with the conditions of the original offer. These new alternative offers will be blended with the ones the user already stores for the task. As long as there are suitable alternative offers, the user will not send any new REQUEST_FOR_OFFERS message.

Fig. 4 summarizes the main architectural elements presented in this section and their interactions as part of the market-oriented grid architecture proposed.

4. Simulations results

This section studies the best strategies for the user, and also two features of the system: adaptability to load changes and fairness.

Table 1
Catalog of VM types available.

VM type	PEs ($V_j^C \times V_j^S$)	Mem (V_j^m) (GBs)	Disk (V_j^d) (GBs)
Normal	1 PE at 1 GHz	1.5	160
Large	4 PEs at 1 GHz	7.5	850
Extra-large	8 PEs at 1.5 GHz	15	1690

4.1. Cloud provider setup

As explained in Section 2, each cloud provider has a catalog of types of VMs that it can instantiate to attend users requests. For the experiments presented here, providers are assigned a catalog defining three types of VMs that can be instantiated (these types closely correspond to the ones defined in EC2 catalog[5]). This catalog is described in Table 1. These types correspond to the set $\{V_1, \ldots, V_n\}$ introduced in Section 3.3.1. The V_j^{COST} and V_j^{START} parameters for each type are set to minimal values so they do not interfere in the results outcome. Thus, the cost is set to 0, although in real settings administrators could choose to discourage the usage of certain VM types by assigning them higher prices. Also, the creation time is set to 1, which the authors know is fairly optimistic but will not introduce biases in the results: the goal is not to study which is the best/most chosen VM type, but the performance of the system as a whole.

Tasks processing requirements will be expressed in MIs, so we need to convert GHzs to MIPS. Such conversion is never accurate in any architecture, and it strongly depends on the software being run. But an approximate conversion can be 1 GHz = 6000 MIPS.[6]

Also, it is needed to define the amount of hardware resources of each cloud provider. Table 2 shows the amount of physical hosts and the resources of each one: memory, CPUs (with their processing speed) and disks (with their size). The hardware configuration of a standard cluster host is close to typical blade hardware settings,[7] the configuration of hosts in the other cluster types are defined taken that one as reference.

Each provider updates the prices of resources every 50 s. All providers set initial prices as follows: $P_c = 100$ (per MI), $P_m = 1000$ (per MB), $P_c = 1000$ (per GB). Also, as commented in Section 3.3.1, each provider will check for *idle* VMs every 50 s. When a VM is found that has been idle for more than 600 s, the VM is turn off.

4.2. Nodes setup

Nodes in the system (DIET nodes, TAMs, the Virtual Bank) have all the same bandwidth, 1 Mb (which is quite conservative). All messages are 1 Kb. The Virtual Bank retrieves money from providers and splits it among users every 1000 s. We assume messages processing time is negligible. This can be safely assumed even for messages that imply the computation of allocations for a task, as the process has little complexity and this complexity grows linearly with the amount of present VMs and the cluster size.

4.3. Users behavior

We deem interesting to study which strategy is better suited for the user benefit before further research. That way, we can make a reasonable assumption about how users will behave in real situations, which we will apply in our later experiments.

The setting applied to study users strategies is as follows. We assume a scenario with two private clouds, each one getting resources from a *Small* cluster (see Table 2). Also, we assume 20 users, each one with 500 tasks to run. Time between the issuing of new tasks follows an exponential distribution. Initially, the average time between tasks is set to $\lambda^{-1} = 30$ s.

For each task i, its size c_i also follows an exponential distribution with average size 10^6 MI. Also, for each task it is necessary to know the maximum amount of time the user will accept to wait to get the task result. This time will be proportional to the task size and a new magnitude that we denote *urgency factor* f_i. This magnitude simulates the fact that not all results are equally critical for the user, so more important ones will get a higher f_i value. Then, if task i is created at t then the task deadline will be $T_i = t + c_i \times f_i$, that is, the time the user is ready to wait to obtain the result is proportional to both the size and importance of the task. In our experiments f_i is uniformly chosen from the following values: {0.001, 0.01, 0.1, 10, 100}. The memory m_i and disk d_i required are also uniformly chosen from different sets of values. In our experiments these were {10, 20, 30, 40, 50} MBs for memory size and {10, 20, 30, 50, 60, 100} GBs for disk size.

Finally, each task i importance (I_i), which is required to know its priority against other tasks (see Section 3.5.2), must be computed too. As in [16], we split tasks into two categories: *high importance tasks* and *low importance tasks*. Also as in [16], 20% of tasks will be of high importance. The importance of tasks of high importance follows a normal distribution with mean 100 and standard deviation 50. The importance of tasks of low importance follows a normal distribution with mean 10 and standard deviation 5. Note that we do not relate importance with the maximum amount of currency the user will accept to pay for a task. As long as one offer's cost is not greater than the present user budget (minus the cost of the tasks already under execution, to ensure that the user never runs out of enough currency to pay an executed task), the offer can be accepted by the user.

The utility function u applied by users to choose the best offer is:

$$u(A^{\text{COST}}, A^{\text{TIME}}) = (A^{\text{COST}} \times A^{\text{TIME}})^{-1}. \quad (6)$$

The initial price of processing one MI is 100. The initial price of one MB and of one GB of disk is the same, 10 000. Each user has an initial budget of 10^9 currency units.

As explained in Section 3.5.2 users can follow two different strategies:

- Retry asking for offers for those tasks that do get an acceptable offer. Those tasks are stored in a queue ordered by priority. Each user will pick the first task in the queue and send a `REQUEST_FOR_OFFERS` message for that task every time the result of another task is received, and periodically at a certain rate. Experimentally we have seen that a low rate is enough to ensure that stored tasks do not have to wait long periods of time. We set this rate to 500 s.
- Keep alternative offers sent in the `OFFERS` message, i.e. those offers that were not chosen initially by the user. If the grid replies in the `RUN_RESULT` message that it failed to run the task, the user will check first whether there are still alternative offers for that task. If so, one of them will be chosen (using the user' utility function). Only when the user runs out of alternative offers a new `REQUEST_FOR_OFFERS` message will be sent.

Four sets of five experiments were run, each set corresponding to a different users' strategy. Results are shown in Fig. 5 in four set of histograms, one histogram per experiment. Each histogram depicts the amount of tasks that were successful, that did not find a suitable offer due to budget limitations, etc. If we look at the first set of histograms, we see that the proportion of failed tasks is really high (due also in part to the high load). Almost all failed tasks are due to budget constraints: the user cannot afford paying

[5] http://aws.amazon.com/ec2/instance-types/.

[6] See for example the values shown in http://en.wikipedia.org/wiki/Instructions_per_second.

[7] See for example http://www.sgi.com/products/servers/half_depth/2u_intel_2p.html.

Table 2
Hw configurations for cloud providers in experiments.

Cluster	Hosts	CPUs/host ($H_k^C \times H_k^s$)	Mem/host (H_k^m) (GBs)	Disks/host ($H_k^D \times H_k^d$)
Small	4	2 CPUs at 2 GHzs	16	2×1 TBs
Standard	5	8 CPUs at 2 GHzs	64	8×2 TBs
Powerful	10	12 CPUs at 3 GHzs	96	12×2 TBs

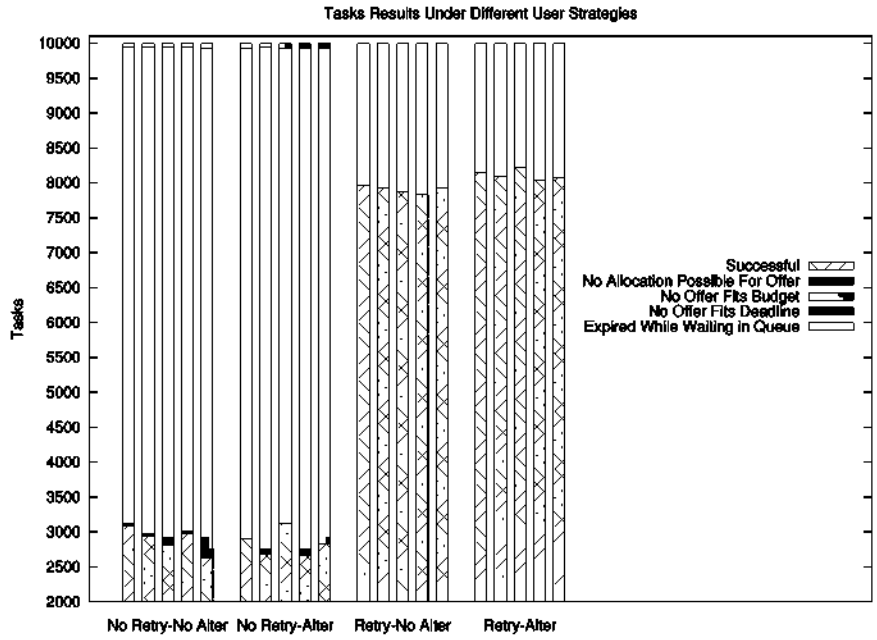


Fig. 5. Impact of retrials on request for offers and usage of alternative offers.

for the task given the offers received ("*No Offer Fits Budget*"). A small set of tasks fail because no offer can run the task before the deadline is met ("*No Offer Fits Deadline*"). And another set of tasks fail because when the cloud provider is asked to run a task under certain cost and time conditions (extracted from the offer chosen by the user) those conditions cannot be fulfilled any more ("*No Allocation Possible For Offer*").

The second set of histograms show the results when the user applies the alternative offers. This policy does not bring any significant improvement in terms of the successful tasks rate.

Much more useful is asking for new offers (i.e. as long as the task deadline can be met) as shown in the third and fourth group of histograms. Now each task is stored until either a provider runs it or the task expires. Using alternative offers slightly improves the rate of successful tasks, i.e., it is better to use alternative offers before performing requests for new ones. Another interesting metric to study is the sum of the values (importance) assigned to the executed and failed tasks, $\sum_{\texttt{Success}} I_i$ and $\sum_{\texttt{Failed}} I_i$, which should be maximized and minimized respectively. In both cases, the combination of using alternative offers and asking for new ones get the best results.

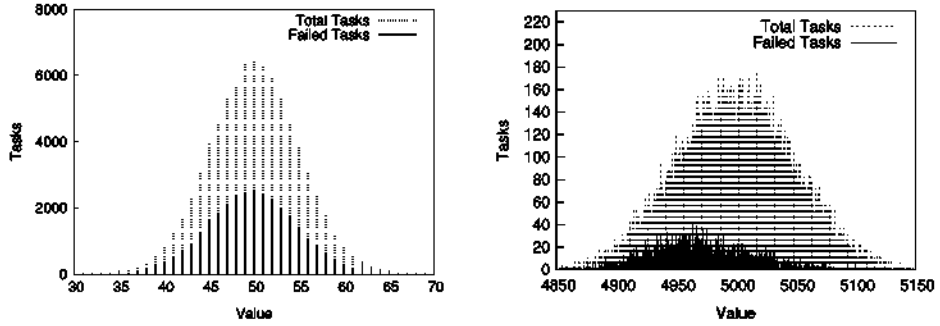
Our results show how simple user strategies such as storing tasks with no offers to retry them later have a significant positive effect on the final system outcome. Thus, it should be assumed that users will implement such strategies in real world situations. In the rest of the simulations presented users will use alternative offers if there are any. When no alternatives offer are available, then the user will store the task in her queue and resend `REQUEST_FOR_OFFERS` messages when the task is chosen again to be executed among the enqueued ones. This contrasts with typical approaches where failed tasks are simply discarded.

4.3.1. Tasks priority

Also, we have studied the positive impact of the prioritization mechanism for stored tasks we propose (see Section 3.5.2), *Priority by Risk*, compared with the most straightforward *Priority by Importance* approach. In these experiments users will retry failed tasks and will use alternative offers. The setting of all parameters is similar to the one used in the previous experiments, but each user will run 5000 tasks, and load is changed by setting an average time between tasks of 50.

Fig. 6 shows the results. Recall that the priority is represented by the value assigned to the task, and that tasks can be of two kinds, those with high value and those with low value (see Section 4.3). Fig. 6(a) and (b) show the amount of failed and total tasks for both types, when users apply priority by importance. Fig. 6(c) and (d) show the results when users order tasks by risk.

It can be observed from Fig. 6 that, for both sets of values, when applying priority by risk the proportion of successful tasks is greater (around 77% in total) that when applying priority by importance (when is only around 66%). Regarding the total value of the successful tasks $\left(\sum_{\texttt{Success}} I_i\right)$, priority by risk yields an improvement of 11% compared with priority by importance. On the other hand $\sum_{\texttt{Failed}} I_i$ is 62% lower when using priority by risk than when using priority by importance. Due to its better performance, *it can be assumed that users will prefer using the priority by risk strategy to order their tasks*. This will be assumed during the next experiments. Also, this was the policy applied in the experiments shown in previous Section 4.3 (we tested that using priority by importance does not alter the conclusion that retrying tasks and using alternative offers is the best choice).



(a) By importance-low value tasks.

(b) By importance-high value tasks.

(c) By risk-low value tasks.

(d) By risk-high value tasks.

Fig. 6. Failed and total tasks for different task priority mechanisms.

4.3.2. About users behavior: summing up

The main goal of this section was to find out the strategies that bring the best outcome for users:

- When no suitable offer is found for some task, it is worth to store it for later retrieval instead of just discarding it, even it this strategy means that tasks will have to "compete" as user' budget is limited.
- It is better to use alternative offers before requesting new offers to the grid.
- The *Priority by Risk* strategy to order enqueued tasks results in less failed tasks.

Once these best strategies have been identified we can build a representative characterization of users. This is necessary to simulate market-based scenarios realistically, where users take decisions regarding tasks ordering, etc., instead of just discarding failed tasks.

4.4. System adaptability

Once the most beneficial/likely strategies for users have been settled, it is time to study the behavior of the market-based system proposed. Two properties must be analyzed: *adaptability* and *fairness*. This section addresses the ability of a cloud provider to adapt to a changing load, while fairness is studied in the next section.

Recall that load can be controlled by setting the average time between tasks for each user, λ^{-1}, to different values. To check system adaptation an experiment will be run where λ^{-1} will be changed to check the performance under different loads. Thus, λ^{-1} is set initially ($t = 0$) to 75, to 7.5 at $t = 10\,000$, to 75 again at $t = 15\,000$ and finally to 750 at $t = 20\,000$. There will be 10 users, each one with 2000 tasks to run, and one single cloud provider on top of a *Powerful* cluster (see Table 2). The rest of the setup is similar to the previous experiments.

Results are shown in Fig. 7. It depicts the amount of allocated and running PEs, along with the number of tasks waiting in VMs queues (the number of tasks in execution is of course equal to the number of running PEs). It can be observed that the system successfully reacts to the increased demand of resources by allocating new Processing Elements in new VMs at $t = 10\,000$, where tasks will be run. Likewise, when the rate is decreased again at $t = 15\,000$ to the initial value the amount of running PEs falls abruptly, and so does the amount of PEs allocated later. Finally, when the rate shrinks at $t = 20\,000$ once again the system adapts and uses a minimum amount of resources. The reason because the changes on the amount of allocated PEs is abrupt is that users choose offers that will cause their tasks to be run in VMs of *Extra-large* type, which are faster (see the VM definitions in Table 1), but require more CPU resources.

4.5. Fairness

Achieving *fairness* is the main goal of grid market-based systems. Fairness refers to how resource usage is split among users by providers. No user should be able to require resources without limits, as this could lead to resource shortage for others. But users should be able to run their tasks as long as they do not impact on other users throughput even if they have a higher resource demand. On the other hand, there should also be a limit on resources usage so when under high demand from several users, those users demanding too many resources will not be able to get all of them.

To test the fairness of our system three experiments were run, each one assigning to the (only) provider cloud available one *Small* cluster, one *Standard* cluster and one *Powerful* cluster respectively. All experiments have 30 users, split into three sets of 10 users each. The average time between tasks for each set are 500, 200, and 25 s respectively. Each user will try to run 2000 tasks.

Fig. 8 shows the amount of tasks successfully run or failed (they expired before they could be executed) during the initial part of the experiment for three users, each one with a different task generation rate (users with the same rate show all very similar

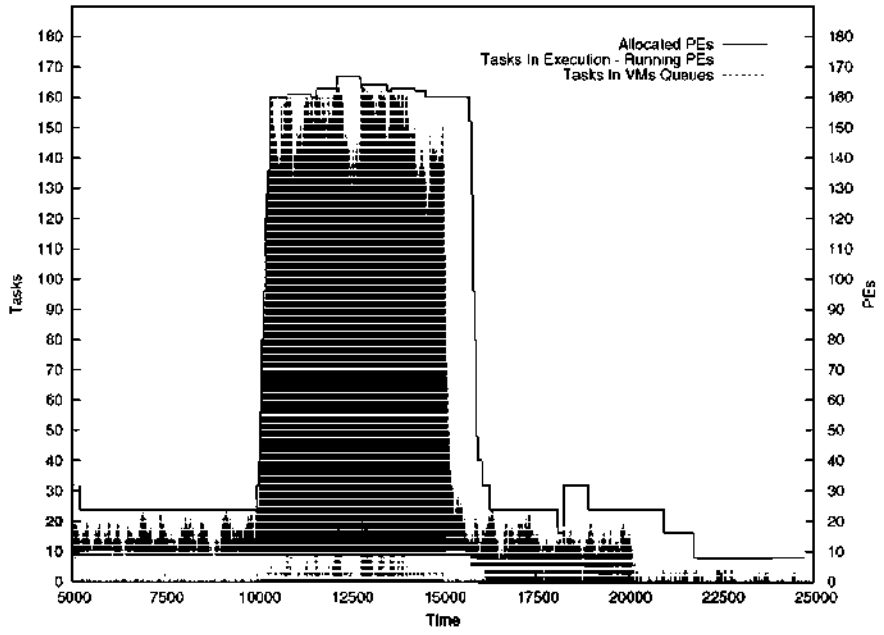
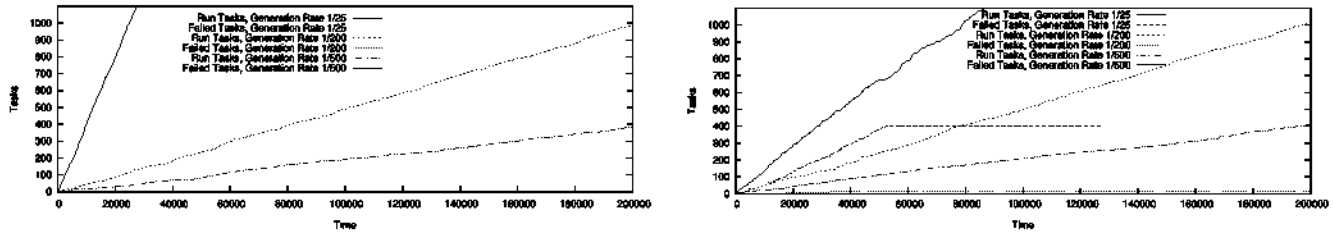
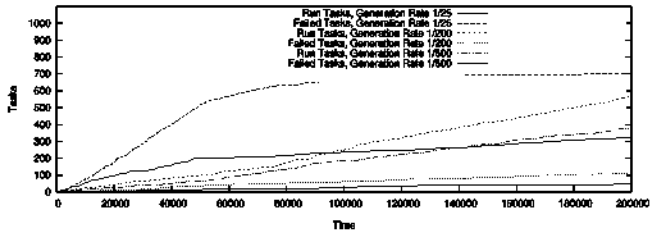


Fig. 7. Adaptability: allocated PEs under varying load.



(a) Powerful cluster.

(b) Standard cluster.



(c) Small cluster.

Fig. 8. Fairness: tasks run and failed, initial part of the experiment.

behavior), in the three different settings. In a powerful cluster (Fig. 8(a)) all users can run their tasks with no restriction, as there are enough resources to attend all petitions regardless of the resources they demand. But in a standard cluster (Fig. 8(b)) the cloud cannot serve all petitions, i.e. there is a certain resources shortage, and so some tasks fail. Yet, this does not affect all users evenly: users with low demand are not affected by this resource shortage and can run their tasks as in the previous setting. Also, users with medium demand are able to run almost all their tasks as before, and are only very lightly impacted by the lack of resources. Users with a high load, however, cannot run all their tasks anymore as they demand more resources than the amount the cloud will grant to any user. As a result, many tasks from users with high load will fail. Note that around $t = 50\,000$ users with high load will have already initiated all their tasks, so from that moment on they will only request to run the tasks enqueued. Finally, when using a small cluster (Fig. 8(c)), the same effect seen in the standard cluster is found again but amplified. Users with small and medium load are only lightly affected, as their demand for resources can be attended by the cloud. In contrast, many tasks from users with a high generation rate fail (more in fact that the amount of run tasks). Note also how the rate of successful tasks from the users with medium rate is increased little after $t = 50\,000$. The reason is that users with high rate are only trying to run tasks stored in their queues, thus effectively lowering their need for resources.

In Fig. 9 we show the run and failed tasks for the small cluster until the end of the experiment. While users with low and medium demand keep the same task execution rate, users with high demand only very slowly are able to keep running tasks. After users with medium load have left the system (they have finished all their tasks) the demand of resources is so low that users can again run requests at high rate (recall that users check their tasks stored in their queues every 500 s and also every time that one task result is received, which allows for fast re-sending of tasks when resources are available).

These results lead to interesting conclusions. Users can try to run more tasks up to a certain rate as long as they do not interfere

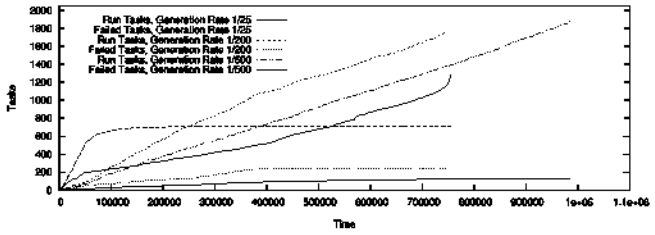


Fig. 9. Fairness: tasks run and failed in small cluster.

with other users. This is positive, as we do not force all users to work at the lowest rate. But if the resource demand by some user is too high then the system penalizes the user by not running many of her tasks, the user will have to store them until many eventually expire (fail), the system does not subtract resources from other users needs.

5. Related work

Despite being a recent technology, cloud computing has already raised the interest of the research community. Present research on IaaS systems is focusing on two main topics:

- Enabling the allocation of distributed resources on federated cloud systems. Open Cirrus [17], the *Sky Computing* [11] initiative or the EU Reservoir [18] joint research project are works oriented to the construction of such environments.
- Automatic scaling to adjust resources allocation to the demand. Automatic scaling is already implemented in some commercial solutions such as Amazon, where users configure scaling actions based on hardware state metrics such as CPU usage, etc. Other works [19,20] propose more flexible scaling mechanisms based on service state in federated environments.

Regarding clouds and grids, there was some initial confusion about the differences and similarities between the two, although this was soon addressed by the community [1]. Later work [21] has further clarified the distinction between them, and analyzed how grids could evolve to benefit from the ideas introduced by the cloud (or the other way around, see for example [22]). In [23,24] the authors present an architecture for the dynamic provision of resources to the virtual organizations (VO) of a grid. Part of the same team lead the StratusLab[8] project, a strong initiative in this regard. StratusLab is an EU joint research project that views clouds and grids as complementary technologies. StratusLab proposes three methods to integrate them:

- Deploy a grid site (based on EGEE[9] software) within a public cloud (Amazon's).
- Apply clouds for resource provisioning in grids.
- Add IaaS-like interfaces to existing grid services.

The second method lies close to the approach introduced in this work. But StratusLab goal is to virtualize an entire grid site for dynamic provision of worker nodes, while this proposal rather connects a grid system (DIET) to one or more clouds to get a supply of VMs in the same dynamic mode. StratusLab, on the other hand, does not apply economic models to ensure fair resource sharing.

5.1. Economy-based grid systems

Applying an economic approach in a grid system is hardly a new idea. Buyya et al. [4] already introduced a market based framework for grids, with an analysis of different market approaches such as auctions, posted price, tendering/contract-net, etc. In [25] the authors further discuss how economy can be applied to efficiently manage resources on grid environments and the advantages of such solutions (automatic regulation of supply and demand, scalability...).

It is not our intention to make a complete survey of all economy-based grid proposals (see [13,5] for such overview of market-oriented grid systems). But in the remaining of this section we will comment how some of those works relate to two main aspects of the system proposed here, i.e. price computation and currency distribution.

5.1.1. Price computation

Regarding proposals for resource price computation, Libra [26] and Libra+$ [16] suggest mechanisms for setting resource prices depending on demand. However they depend on some parameters whose values are arbitrary (must be tuned depending on the system and tasks). In contrast, our pricing solution does not requires such parameter values guessing.

G-commerce [27] proposes a formal pricing solution based on markets theory that aims to get the *equilibrium* prices of all resources. The equilibrium price is a market concept. In a market scenario, if the price of a commodity is low the demand will grow, in turn if the price of a commodity is high the demand will decrease. The equilibrium price in a market is the price reached when supply is equal to the demand. Unfortunately, such solution cannot be applied here. To compute the equilibrium price is required to have knowledge of the *global* demand of all resources in all SeDs, which we assume is not feasible in many scenarios.

On the other hand auction systems such as Bellagio [28], Mirage [14], and Tycoon [29] do not need providers to compute the price of resources. Users are the ones who must compete for the resources they require by bidding, so the resource is assigned to the highest bids. But then is the users who must decide policies to set the initial bid, how much increase the bid each time, what is the maximum bid, etc. So an auction approach is not easier to implement, it simply assigns more responsibility to users.

Other proposals such as FirstPrice [30], FirstReward [31] or FirstOpportunity [32] do not propose any resource price computation mechanism.

5.1.2. Distribution of virtual currency

Currency creation and circulation is an important concept in any market system. An option suggested in some works is to use real currency instead of virtual one [28], so users will take real care when demanding resources. Also, this solution frees the system from having to inject virtual currency and assign it to users. However, this approach has several inconveniences. For example, users with more economic means (e.g. better funding) will get more resources, leading to unfair situations. Also, in scientific environments users could be reluctant to spare real currency for resources as they often work in other kind of settings were resources, even if scarce, are freely available. Thus, using *virtual currency*, created and distributed by the system seems to be the most feasible solution.

There are several options to inject and circulate virtual currency:

- Each provider (SeD) periodically reports to some central entity (Virtual Bank) about the value of all the resources it can deliver, taking into account their updated price. This would represent the 'wealth' of the system. This amount is then split and send to the users. However this solution would cause permanent inflation.
- The Virtual Bank periodically sends a certain amount to all users. Providers do not hoard money. But it is then necessary to

[8] http://stratuslab.eu.

[9] http://www.eu-egee.org.

decide how much assign to users, i.e. how much currency inject to the system. Arbitrary amounts could cause artificial inflation or deflation.

- The SeD/Cloud does not hoard neither drops the currency it gathers. Instead, it sends it to the Virtual Bank which will forward it to the users. This approach seems the more suitable.

In fact, the third approach is similar to the idea proposed in Mirage [14]. Also, to avoid hoarding by users, Mirage implements a *taxing* system that periodically reduces users budget so they do not tend to store currency too long. The currency obtained thorough this taxing system is then distributed back to user, as in the case of the clouds income. A similar mechanism could be used in our proposal.

Other works do not shed light to this problem, at least in the scenario proposed here. In Bellagio [28] users receive a budget proportional to the resources they provide, but this cannot be applied here as for the sake of flexibility DIET users are not assumed to be providers as well (although they could be). G-commerce [27] follows a similar approach to the one defined in the second point of the list above. They do not set any mechanism to decide how much to assign to users at each iteration. Tycoon [29] does not make any assumption, users *"...are funded at some regular rate. The system administrators set their income rate based on exogenously determined priorities"*, or *"...bring resources ... must earn funds by enticing other users to pay for their resources"*. FirstPrice [30] does not say anything about the subject. FirstReward [31] proponents explicitly state that they do not address how currency is injected or recycled. FirstProfit, FirstOpportunity [32] and Aggregate Utility [33] do not say anything about the subject (Aggregate Utility [33] in fact encourages using real currency).

6. Conclusions

The system presented in this paper is a proposal to combine grid and cloud systems through a market-based approach. Grids can benefit from clouds by requesting and releasing resources from them, thus not being forced to have their own pool of resources. However, the grid system needs some criteria to know when to take resource scaling decisions. This criteria must of course take into account the demand induced by the tasks sent by the users.

By applying the pricing adaptation mechanism here proposed, grids can now scale resources automatically, while at the same time ensuring fairness in resource sharing. Future work will consist on implementing this on a real system: adapting DIET, programming the Virtual Bank and TAM, and connecting the TAM to some IaaS cloud provider, based for example on OpenNebula.

Acknowledgments

Dr. Luis Rodero-Merino was financially supported first by INRIA and then by UPM during this research. This work is partially supported by ANR MapReduce project (ANR-10-SEGI-001-02), by the EC under project CumuloNimbo FP7-257993, by the Madrid Research Council (CAM) under project CLOUDS S2009TIC-1692 and by the Spanish Science Foundation under project CloudStorm TIN2010-19077.

Appendix. Extensions to GridSim

Our simulation software is based on GridSim [6], a well known simulation framework in Java for grid environments. In this section we briefly introduce the design of our simulator and the extensions done to the main GridSim components, needed for our cloud-grid hybrid system. Keep in mind that the diagrams shown in this section are merely descriptive, and do not provide a complete design view of the simulation software.

Generic GridSim simulations are based on the `GridSimCore` class. This class has the basic functionality for net communications

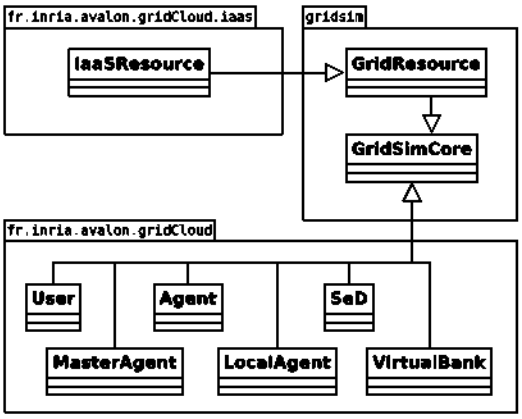


Fig. A.1. Main simulation entities.

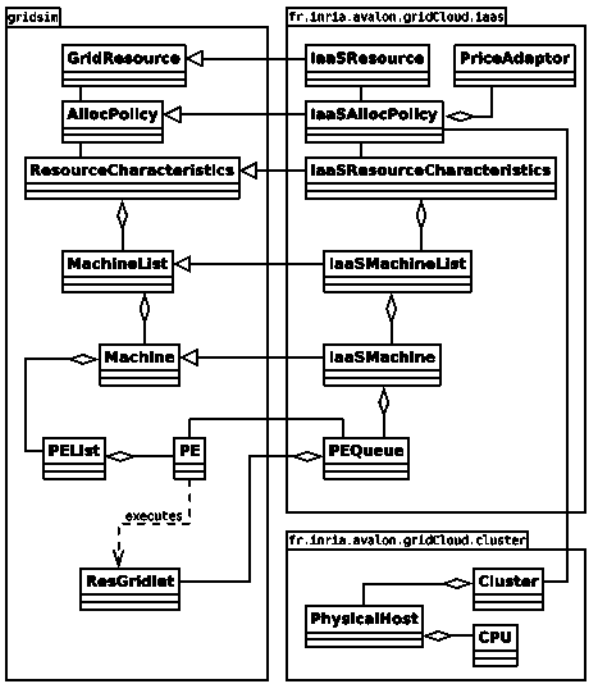


Fig. A.2. Cloud provider components.

and event handling. The DIET-related entities of our simulations, i.e. `User`, `MasterAgent`, `Agent` and `LocalAgent` extend this class as shown in Fig. A.1. The `VirtualBank`, which gathers the currency earned by the cloud providers and sends it back to the users (see Section 3.1) is based on this class as well. These classes use default GridSim functionality, without adding new features to its standard behavior. The `GridResource` class of GridSim, on the other hand, is intended to represent a group of machines that are available for the grid tasks. But it is not designed to support the dynamic addition or removal of machines (or VMs in this case). We extend it with the `IaaSResource` class, that extends the default behavior so the VMs available can be changed as the simulation evolves. Each IaaS cloud will be represented by an instance of `IaaSResource`.

The mechanism to compute allocations and assign tasks to PEs, which is described in Section 3.3.2, is implemented by class `IaaSAllocPolicy`, which extends GridSim's `AllocPolicy` class. Each instance of `IaaSAllocPolicy` will contain an instance of the `PriceAdaptor` class for price adaptation. Thus, TAM's functionality is implemented by `IaaSAllocPolicy`.

To simulate the management of tasks, VMs, and physical hosts, it was necessary to extend several GridSim classes, as shown

in Fig. A.2. In GridSim, the `Machine` class represents a host, containing a set of PEs (class PE). Each PE instance executes tasks, which are represented by the `ResGridlet` class. A new `IaaSMachine` class, based on `Machine` has been developed. Each instance of this class represents a VM. This class handles a queue of tasks per PE, which is not provided by GridSim. Physical resources are represented in a new package `cluster`. Accounting of physical resources (as explained in Section 3.3.2) is performed by this package.

References

[1] Luis M. Vaquero, Luis Rodero-Merino, Juan Cáceres, Maik Lindner, A break in the clouds: towards a cloud definition, ACM SIGCOMM Computer Communication Review 39 (2009) 50–55.

[2] Luis M. Vaquero, Luis Rodero-Merino, Rajkumar Buyya, Dynamically scaling applications in the cloud, ACM SIGCOMM Computer Communication Review 41 (2011) 45–52.

[3] Eddy Caron, Frédéric Desprez, Adrian Muresan, Forecasting for grid and cloud computing on-demand resources based on pattern matching, in: IEEE, editor, Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2010, Indianapolis, Indiana, USA, November 2010. To appear as work in progress paper track.

[4] Rajkumar Buyya, David Abramson, Jonathan Giddy, Heinz Stockinger, Economic models for resource management and scheduling in grid computing, Concurrency and Computation: Practice and Experience 14 (2002) 1507–1542.

[5] Aminul Haque, Saadat M. Alhashmi, Rajendran Parthiban, A survey of economic models in grid computing, Future Generation Computer Systems 27 (2011) 1056–1069.

[6] Rajkumar Buyya, Manzor Murshed, GridSim: a toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing, Concurrency and Computation: Practice and Experience 14 (2002) 1175–1220.

[7] Eddy Caron, Frédéric Desprez, DIET: a scalable toolbox to build network enabled servers on the grid, High Performance Computing Applications 20 (2006) 335–352.

[8] Eddy Caron, Frédéric Desprez, David Loureiro, Adrian Muresan, Cloud computing resource management through a grid middleware: a case study with DIET and Eucalyptus, in: Proceedings of the 2nd IEEE International Conference on Cloud Computing, Cloud'09, IEEE, Bangalore, India, 2009, pp. 151–154.

[9] Rafael Moreno-Vozmediano, Rubén S. Montero, Ignacio M. Llorente, Elastic management of cluster-based services in the cloud, in: Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds, ICAC'09, ACM, Barcelona, Spain, 2009, pp. 19–24.

[10] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, Dmitrii Zagorodnov, The Eucalyptus open-source cloud-computing system, in: Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID'09, IEEE, Shanghai, China, 2009, pp. 124–131.

[11] Katarzyna Keahey, Maurício Tsugawa, Andréa Matsunaga, José A.B. Fortes, Sky computing, Computer 13 (2009) 43–51.

[12] Jeffrey Shneidman, Chaki Ng, David C. Parkes, Alvin AuYoung, Alex C. Snoeren, Amin Vahdat, Brent Chun, Why markets could (but dont currently) solve resource allocation problems in system, in: Proceedings of the 10th Workshop on Hot Topics in Operating Systems, HotOS'05, USENIX, Santa Fe, USA, vol. 10, June 2005.

[13] James Broberg, Srikumar Venugopal, Rajkumar Buyya, Market-oriented grids and utility computing: the state-of-the-art and future directions, Journal of Grid Computing 6 (3) (2008) 255–276.

[14] B.N. Chun, P. Buonadonna, A. AuYoung, Chaki Ng, D.C. Parkes, J. Shneidman, A.C. Snoeren, A. Vahdat, Mirage: a microeconomic resource allocation system for sensornet testbeds, in: Proceedings of the 2nd IEEE Workshop on Embedded Networked Sensors, EmNetS05, IEEE, 2005, pp. 19–28.

[15] Lusajo M. Minga, Yu-Qiang Feng, Yi-Jun Li, Dynamic pricing: ecommerce-oriented price setting algorithm, in: Proceedings of the 2nd International Conference on Machine Learning and Cibernetics, ICMLC 2003, vol. 2, November 2003, pp. 893–898.

[16] Chee Shin Yeo, Rajkumar Buyya, Pricing for utility-driven resource management and allocation in clusters, International Journal of High Performance Computing Applications 21 (4) (2007) 405–418.

[17] Arutyun I. Avetisyan, Roy Campbell, Indranil Gupta, Michael T. Heath, Steven Y. Ko, Gregory R. Ganger, Michael A. Kozuch, David OHallaron, Marcel Kunze, Thomas T. Kwan, Kevin Lai, Martha Lyons, Dejan S. Milojicic, Hing Yan Lee, Yeng Chai Soh, Ng Kwang Ming, Jing-Yuan Luke, Han Namgoong, Open cirrus: a global cloud computing testbed, Computer 43 (2010) 35–43.

[18] Benny Rochwerger, David Breitgand, Eliezer Levy, Alex Galis, Kenneth Nagin, Ignacio M. Llorente, Rubén Montero, Yaron Wolfsthal, Erik Elmroth, Juan Cáceres, Muli Ben-Yehuda, Wolgang Emmerich, Fermín Galán, The Reservoir model and architecture for open federated cloud computing, IBM Journal of Research and Development 53 (4) (2009) 1–11.

[19] Luis Rodero-Merino, Luis M. Vaquero, Víctor Gil, Javier Fontán, Fermín Galán, Rubén S. Montero, Ignacio M. Llorente, From infrastructure delivery to service management in clouds, Future Generation Computer Systems 26 (2010) 1226–1240.

[20] Rajkumar Buyya, Rajiv Ranjan, Rodrigo N. Calheiros, InterCloud: utility-oriented federation of cloud computing environments for scaling of application services, in: Proceedings of the 10th International Conference on Algorithms and Architectures for Parallel Processing, ICA3PP 2010, in: Lecture Notes in Computer Science, vol. 6081, Springer, Busan, South Korea, 2010, pp. 13–31.

[21] Shantenu Jha, Andre Merzky, Geoffrey Fox, Using clouds to provide grids with higher levels of abstraction and explicit support for usage modes, Concurrency and Computation: Practice and Experience 21 (2009) 1087–1108.

[22] Michael A. Murphy, Sebastien Goasguen, Virtual organization clusters: self-provisioned clouds on the grid, Future Generation Computer Systems 26 (2010) 1271–1281.

[23] Manuel Rodríguez, Daniel Tapiador, Javier Fontán, Eduardo Huedo, Rubén S. Montero, Ignacio M. Llorente, Dynamic provisioning of virtual clusters for grid computing, in: Proceedings of Euro-Par 2008 Workshops, 3rd Workshop on Virtualization in High-Performance Cluster and Grid Computing, VHPC'08, in: Lecture Notes in Computer Science, vol. 5415, Springer, Las Palmas de Gran Canaria, Canarias, Spain, 2008, pp. 23–32.

[24] Constantino Vázquez, Eduardo Huedoa, Rubén S. Montero, Ignacio M. Llorente, On the use of clouds for grid resource provisioning, Future Generation Computer Systems 27 (2011) 600–605.

[25] Rajkumar Buyya, David Abramson, Srikumar Venugopal, The grid economy, Proceedings of the IEEE 93 (2005) 698–714.

[26] Jahanzeb Sherwani, Nosheen Ali, Nausheen Lotia, Zahra Hayat, Rajkumar Buyya, Libra: a computational economy-based job scheduling system for clusters, Software Practice and Experience 34 (6) (2004) 573–590.

[27] Rich Wolski, James S. Plank, Todd Bryan, John Brevik, G-commerce: market formulations controlling resource allocation on the computational grid, in: Proceedings of the 15th International Parallel and Distributed Processing Symposium, IPDPS2001, IEEE, 2007.

[28] Alvin Auyoung, Brent N. Chun, Alex C. Snoeren, Amin Vahdat, Resource allocation in federated distributed computing infrastructures, in: Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure, OASIS04, October 2004.

[29] Kevin Lai, Lars Rasmusson, Eytan Adar, Li Zhang, Bernardo A. Huberman, Tycoon: an implementation of a distributed, market-based resource allocation system, Multiagent and Grid Systems 1 (3) (2005) 169–182.

[30] Brent N. Chun, David E. Culler, User-centric performance analysis of market-based cluster batch schedulers, in: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGrid02, IEEE, 2002, p. 30.

[31] David E. Irwin, Laura E. Grit, Jeffrey S. Chase, Balancing risk and reward in a market-based task service, in: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing, HPDC04, IEEE, 2004, pp. 160–169.

[32] Florentina I. Popovici, John Wilkes, Profitable services in an uncertain world, in: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, SC05, IEEE, 2005, p. 36.

[33] Alvin AuYoung, Laura Grit, Janet Wiener, John Wilkes, Service contracts and aggregate utility functions, in: Proceedings of the 2006 15th IEEE International Conference on High Performance Distributed Computing, HPDC06, IEEE, 2006, pp. 119–131.

L. Rodero-Merino is a postdoctoral Researcher at INRIA (Avalon group, Laboratoire de l'Informatique du Parallélisme) with Telefónica ICD. He graduated in Computer Science at the University of Valladolid and received his Ph.D. degree at University Rey Juan Carlos, where he also worked as an Assistant Professor. Previously he had worked in the R&D area of Ericsson Spain. He was a researcher at Telefónica R&D before joining INRIA. His research interests include computer networks, distributed systems, P2P systems, grid computing and cloud computing. See http://sites.google.com/site/luisroderomerinowebpage for further information.

Eddy Caron is an assistant professor at Ecole Normale Supérieure (Lyon) and holds a position with the GRAAL project (LIP laboratory, ENS). He received his Ph.D. in C.S. from University de Picardie Jules Verne in 2000 and his HDR (Habiliation Diriger les Recherche) from the ENS in 2010. His research interests include scientific computing on parallel distributed memory machines, grid computing and cloud computing. He is involved in many program committees (HCW, ISPA…). He is co-chair of the GridRPC working group in OGF. He is the co-funder and scientific consultant of SysFera (www.sysfera.com). See http://graal.ens-lyon.fr/~ecaron for further information.

Adrian Muresan is a second year Ph.D. student at Ecole Nationale Superior in Lyon, France. His Ph.D. is focused on resource management in Cloud platforms. He has obtained a Masters Degree in Computer Science at the Technical University of Cluj-Napoca, Romania in 2009.

Frédéric Desprez is a director of research at INRIA and holds a position at LIP laboratory (ENS Lyon). He received his Ph.D. in C.S. from the Institut National Polytechnique de Grenoble in 1994 and his MS in C.S. from the ENS Lyon in 1990. His research interests include parallel algorithms, scheduling for large scale distributed platforms, data management, and grid and cloud computing. See http://graal.enslyon.fr/~desprez/ for further information.